GestIT: A Declarative and Compositional Framework for Multiplatform Gesture Definition

Lucio Davide Spano
Università di Cagliari
Via Ospedale 72,
09124, Cagliari, Italy
davide.spano@unica.it

Antonio Cisternino
Università di Pisa
Largo B. Pontecorvo 3,
56127, Pisa, Italy
cisterni@di.unipi.it

Fabio Paternò
ISTI-CNR
Via G. Moruzzi 1,
56127, Pisa, Italy
fabio.paterno @isti.cnr.it

Gianni Fenu
Università di Cagliari
Via Ospedale 72,
09124, Cagliari, Italy
fenu @unica.it

ABSTRACT
Gestural interfaces allow complex manipulative interactions that are hardly manageable using traditional event handlers. Indeed, such kind of interaction has longer duration in time than that carried out in form-based user interfaces, and often it is important to provide users with intermediate feedback during the gesture performance. Therefore, the gesture specification code is a mixture of the recognition logic and the feedback definition. This makes it difficult 1) to write maintainable code and 2) reuse the gesture definition in different applications. To overcome these kinds of limitations, the research community has considered declarative approaches for the specification of gesture temporal evolution . In this paper, we discuss the creation of gestural interfaces using GestIT, a framework that allows declarative and compositional definition of gestures for different recognition platforms (e.g. multitouch and full-body), through a set of examples and the comparison with existing approaches

Author Keywords
Gestural interaction, Input and Interaction Technologies, Analysis Methods, Software architecture and engineering, User Interface design.

ACM Classification Keywords
H.5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

INTRODUCTION
The wide availability of devices with multitouch capabilities (phones, tablets and desktops), together with the spread of sensors that are able to track the whole body for interaction (such as Microsoft Kinect, Creative Interactive Gesture Camera, Leap Motion), has enabled a pervasive introduction of gestural interaction in our everyday life.

From the development point of view, the possibility to exploit such new devices for creating more engaging interaction has been built on top of existing User Interface (UI frameworks and their reactive programming models. While this is reasonable from a software reuse point of view, applying the event-based management of the UI behaviour is difficult when dealing with gestural interaction, since a gesture is better represented as something varying over time rather than as an event corresponding to an action on a classical WIMP interface (e.g. a button click). Therefore, the application UI usually needs to provide users with intermediate feedback during the gesture performance, and modelling an entire gesture with a single event forces the developers to redefine the gesture recognition logic.

A possible solution for such kind of problem is provided by declarative and compositional approaches for the gesture definition. In this paper, we discuss the advantages and the drawbacks of this type of approach through our experience with the GestIT framework.

CONTRIBUTION
In this paper, we discuss how it is possible to address a set of problems in the engineering and development of gestural interfaces. The first and the second one are related to the gesture modelling in general, while the third is related to the compositional approach for gesture definition. The three problems we address can be summarized as follows:

1. *It is difficult to model a gesture only with a single event raised when its performance is completed*. The need for intermediate feedback forces the developer to redefine the tracking part. From now on, we refer to this issue as the *granularity problem*.
2. In [11], the authors state "*Multitouch gesture recognition code is split across many location in the source"*. This problem is even worse if we consider full-body gesture recognition, which has a higher number of points to track in addition to the other features (e.g. joints orientation, voice etc.). We refer to this issue as the *spaghetti code problem*.
3. A compositional approach for gestures has to deal with the fact that *"Multiple gestures may be based on the same initiating sequence of events"* [11]. This means that a support for the gesture composition has to manage possible ambiguities in the resulting gesture definition. We refer to this issue as the *selection ambiguity problem*.

EICS'13, June 24–27, 2013, London, United Kingdom.

In this paper, we discuss the advantages of a declarative and compositional approach for gestural interaction, which are able to solve the aforementioned problems. In our examples we use GestIT[1], a supporting framework that allows using the same composition mechanism on different platforms.

The paper is organised as follows: after the discussion of the related work, we compare GestIT with the most complete declarative approach for gesture definition existing in literature [11, 12] and we detail the procedure for creating a gesture-based application through a concrete example. After that, we explain how it is possible to address the problems 1 and 2, leveraging the approach of currently available frameworks, and different solutions for the problem 3. In particular, we discuss the GestIT support for postponing the choice of the actual recognized gesture, offering the possibility to implement the conflicting changes on the UI as long running transactions. Finally, we provide an overview on cross-platform how it is possible to support a cross-platform gesture definition exploiting the discussed approach.

RELATED WORK

Multitouch frameworks

Commercial frameworks for the development of multitouch interfaces (iOS [2], Android [1], etc.) are really similar to each other, and they all manage this kind of interaction at two levels. The first is the availability of common high-level gestures (such as pinch and rotation), which are natively supported by UI controls. The notification is based on a single event at the end of the gesture completion. The second is the possibility to define custom gestures or, as we already explained, to provide intermediate feedback during the gesture performance. This is left to the handling of low-level touch events, which are similar to mouse events.

Full-body frameworks

Different frameworks for the development of full body gestural interfaces exist, and they may be categorized according to the type of devices they support. For depth-camera based devices the most important ones are the Microsoft Kinect SDK [13] and the Primesense NITE framework [16]. Both development kits allow the user's skeleton tracking as a set of joint points, providing the position in 3D and the orientation. The Kinect SDK allows also the tracking of face expressions. NITE instead provides some predefined UI controls that recognize a set of gestures (wave, swipe etc.). The approach for the gesture modelling is the same discussed for the multitouch controls: the controls notify a single event at the completion of the given gesture. SDKs for remote controls provide the access to the low-level data sent by the device, without providing a direct support for the gesture recognition. For instance, WiiLib [19] and WiimoteLib [14] provide the possibility to interpret correctly the bytes sent by the Nintendo Wiimote controller for developing applications.

DECLARATIVE APPROACHES

We group in the declarative approach category the attempts to provide a formal description of the gesture performance. CoGest [7] exploited context-free grammars in order to describe conversational gestures, while GeForMT [9] exploited the same approach for describing multitouch gestures. Neither of them provides gesture recognition capabilities for the defined grammars. Other approaches such as GDML [5] allow the gesture recognition but provide only a single event when the gesture is completed.

The frameworks Midas [17] and Mudra [8] propose a rule-based definition of multitouch gestures and multimodal input. Both of them are affected by the *spaghetti code* problem, since the definition of the interaction effects on the UI is included in the rules specification. GestIT separates the gesture definition from the UI behavior code. In addition, GestIT provides more flexible handling of the ambiguities, since both approaches solve this problem simply forcing the selection of one gesture through priorities.

GISpL [4] suffers both *granularity* and *spaghetti code* problems, since gestures cannot be decomposed into sub-gestures and the behavior is defined together with the feature tracking. The latter problem affects also the approach in [10]. Shwartz et al. [18] provide a solution for the uncertainty only after the gesture performance. Instead, GestIT manages such uncertainty also during the gesture execution, which is crucial for providing intermediate feedback.

In the following sub-sections, we analyse the two approaches that to the best of our knowledge are the only ones that offer a declarative formalization together with recognition capabilities.

Proton++

Proton++ [11, 12] is a multitouch framework allowing developers to declaratively describe custom gestures, separating the temporal sequencing of the events from the code related to the behaviour of the UI. Multitouch gestures are defined as regular expressions, where literals are identified by a triple composed of 1) the event type (touch down, move and up), 2) the touch identifier (e.g. 1 for the first finger, 2 for the second etc.) and 3) the object hit by the touch (e.g. the background, a particular shape etc.).

It is possible to define a custom gesture exploiting the regular expression operators (concatenation, alternation, Kleene star). The underlining framework is able to identify conflicts between different composed gestures and to return their common longer prefix in order to 1) let the developers remove the ambiguous expression or 2) assign different probability scores to the two gestures. The runtime support receives the raw input from the device, transforms it into a

[1] http://gestit.codeplex.com/

touch event stream that is matched against the defined regular expressions.

When one or more gestures are recognized, the support invokes the callbacks associated to the related expressions, selecting those with higher confidence scores (assigned by the developer in case of conflict between the expression definitions at design time). The improved version of the framework (presented in [12]) included also the possibility for the developer to calculate a set of attributes that may be associated to an expression literal. For instance, it is possible to associate the current trajectory to a touch move event, and let the framework raise the associated events (read recognize the literal) only if its movement direction is the one that the designer specified (e.g. north, north-west, south etc.). Other examples of such attributes are the touch shape, the finger orientation etc. In Proton++ it is possible to define the custom gestures through a graphical notation (called tablature), which has been demonstrated to be more understandable for the developers if compared with normal code.

GestIT

GestIT [20] shares with Proton++ the declarative and compositional approach, and it is able to model gestures for different kind of devices such as multitouch screens or full-body tracking devices. A custom gesture in GestIT is defined through an expression, starting from a set of ground terms that can be composed with a set of operators. The operators are based on those provided by CTT for defining temporal relationships in task modelling [15].

Ground terms represent the single features that are tracked by a given recognition device. For instance, if we consider a multitouch screen, the features that are tracked by the device are the position of the touches, while if we consider a full-body tracking device the features are the skeleton joint positions and orientations. In addition, it is possible to associate a predicate to each ground term in order to constraint its recognition to a given condition, which may be computed considering the current and/or the previous device events (e.g. the trajectory, speed etc.). The composition operators are the following:

- *Iterative* (symbol $*$), which recognizes a given gesture an indefinite number of times
- *Sequence* (symbol $\gg$), which expresses a sequence relationships among the operands, which are recognized in order, from left to right.
- *Parallel* (symbol ||), which defines the simultaneous recognition of two or more gestures
- *Choice* (symbol []), which allows the recognition of only one among the connected sub-gestures
- *Disabling* (symbol [>), which stops the recognition of another gesture, typically used for stopping the iteration loops.
- *Order Independence* (symbol |=|), which expresses that the connected sub-gestures may be performed in any order. But when the user starts performing one of the sub-gestures, s/he has to complete it before starting another one.

Comparison

In the next section we demonstrate that the possible gestures modelled using Proton++ are a subset of those that may be defined with GestIT. We prove it showing a general way for mapping Proton++ definition towards the GestIT notation. In addition, we show that there is a class of gestures described by GestIT, which is not possible to define using Proton++. Obviously, since Proton++ describes only multitouch gestures, we define the correspondence between the regular expression literals and the ground terms only for the multitouch platform.

However, it is worth pointing out that the better expressivity of GestIT modelling approach is not due to the multitouch platform domain, but rather to a less expressive set of operators provided by Proton++. Indeed, it is possible to model full-body gestures using the Proton++ approach, providing a set of literals related to a full-body tracking device. Even in this case there is a set of gestures that can be expressed with GestIT but not with Proton++.

Proton++ literals

A Proton++ literal is identified by:

1. An event type (touch down, touch move, touch up)
2. A touch identifier
3. An object hit by the touch
4. A set of custom attributes values (one or more)

In GestIT for multitouch a ground term is identified by an event type (touch start, touch move or end) and by a touch identifier. Therefore the correspondence between the first two elements of the Proton++ literal and the GestIT ground term is straightforward. The third and fourth component of a Proton++ literal can be all modelled constructing a correspondent predicate associated to a GestIT ground term. We recall that a predicate associated to a ground term in GestIT is a boolean condition whether the gesture performance conforms to a set of gesture-specific constraints. According to this definition, the third component can be modelled with a predicate that checks if the current touch position is contained into an object with a given id or belonging to a particular class.

The forth component can be modelled considering, for each Proton++ custom attribute value, the function that computes its value according to the previous and the current position of the touch. Since it has to be defined in Proton++ for being associated to a literal, it is also possible to provide a predicate that compares the current attribute value with the desired one, in order to be used in GestIT. If more than one value is acceptable, the predicate can be defined simply through a boolean OR of the comparison for the different values. Obviously, if both the third and the fourth identification component of the literal need to be modelled,

it is sufficient to define a single predicate that is composed by the boolean AND of the corresponding predicates.

Table 1 summarizes how to transform a Proton++ literal into a GestIT ground term.

Proton:	GestIT
$E_{T_{id}}^{O\|V_1 \dots V_n}$	$E_{T_{id}}[p]$ where: $p = o \wedge (a_1 \vee \dots \vee a_n) \quad i = 1 \dots n$ $o = true \Leftrightarrow O_{type} = O$ $a_i = true \Leftrightarrow A_i = V_i \quad i = 1 \dots n$

Table 1: correspondence between the Proton++ literal and the GestIT ground term. *E* represents an event type, T_{id} a touch identifier. The *o* predicate models the touched object constraint in Proton++ literals: O_{type} is a property that maintains the current object type, *O* is a concrete value for the object type (e.g. start, rectangle etc.). The a_i predicates model the custom attribute part of a Proton++ literal: A_i is a property that maintains the value of the attribute, while V_i is the actual attribute value. The *p* predicate puts all the definitions together and it is associated to the ground term in GestIT.

Proton++ operators

The correspondence between the Proton++ and GestIT operators is straightforward, since for each one defined by the former there is an equivalent in the latter. Table 2 summarizes how to transform the operators from Proton++ to GestIT.

Proton++:	GestIT
Concatenation: $A_P B_P$	Sequence: $A_G \gg B_G$
Alternation: $A_P \| B_P$	Choice: $A_G [\,] B_G$
Kleene star: $A_P{}^*$	Iterative: $A_P{}^*$

Table 2: correspondence between the Proton++ operators and the GestIT ones. A_P, B_P and A_G, B_G represent respectively a Proton++ and a GestIT generic expression.

Appling recursively the transformations defined in Table 1 and Table 2 it is possible to build a GestIT gesture definition corresponding to a Proton++ one.

The vice-versa is not possible in general, since there is no way to transform the *Disabling* and the *Parallel* operators from GestIT to Proton++.

The *Disabling* operator is important in order to stop the recognition of iterative gestures, in particular the composed ones. Most of the times, it models how to interrupt the iterative recognition of a gesture. For instance in a grab gesture, the iterative recognition of hand movements is interrupted by opening the hand.

In addition, it may be used also for modelling situations where the user performs an action that interrupts the interaction with the application. For instance, all the Kinect games have a "pause" gesture disabling the interaction. In the application we describe in the next section, the disable operator is used for modelling the fact that the application tracks the user only if she is in front of the screen. Therefore, the gesture "shoulders not parallel to the screen plane" disables the interaction. This is particularly relevant while interacting with devices continuously tracking the user (e.g. Microsoft Kinect), since it is important to provide the user with a way to disable the interaction at any time.

The *Parallel* operator has a clear impact when modelling parallel input for e.g. multi-user applications. For instance, the parallel operator can be useful in a scenario where a user zooms a photo on a multitouch table while another user drags another picture, simply composing two existing gestures. In addition, it is also possible that parallel interaction occurs with a single user. For instance, a user may drag an object through a single-hand grab gesture and point with the other hand for selecting where to drop it.

DECLARATIVE GESTURE MODELLING

In this section, we show how it is possible to create a gestural interface with GestIT, as a sample for the declarative gesture modelling approach. After that, we show how the framework addresses the granularity, spaghetti code and selection ambiguity problems. In order to facilitate the discussion, we refer to two application examples. The first one is a touchless interface for a recipe browser. It allows the cooker to go through the description of the steps for preparing a dish without touching any device. This is particularly useful while cooking, since the user has dirty hands or is manipulating tools. The second one is a simple 3D model viewer, which can be controlled through gestures. The applications have been already discussed respectively in [21] and [20], here we analyse some of the supported interactions that exemplify the recurrent problems in gesture modelling.

Creating a Declarative Gestural Interface

In this section we detail how a developer can use GestIT in order to create a gestural UI. The application is a touchless recipe browser, organised into three presentations: the first one allows the user to select the recipe type (e.g. starter, first dish, main dish etc.), the second is dedicated to the selection of the recipe, while the last one presents the steps for cooking the selected dish with a video and subtitles.

In the latter presentation it is possible to go through the steps back and further or to randomly jump from one point to the other of the procedure. We consider here the C# version for Windows Presentation Foundation (WPF) of the GestIT library.

An interface in WPF is described by two different files. The first one contains the definition of the UI appearance and layout specified using XAML, an XML-based notation that can be used in .NET applications for initializing objects. In this case, it initializes the widgets contained into the application view. The second file involved in the UI definition contains the behaviour, and it is a normal C#

class file. Since the two files are part of the same view class definition, the latter is called the "code-behind" file. Objects defined by the XAML file are accessible in the code-behind file and the methods defined in the code-behind file are accessible in the XAML definition.

In this example, we discuss the implementation of the first presentation, which is shown in Figure 1. The view is composed of a title on the upper part and a fisheye panel in the centre. The bottom part is dedicated to the status messages: the application notifies if it is tracking the user's movements or not.

Figure 1: Recipe selection UI

The gestural interaction is defined inside the associated view through a set of custom XAML tags, which are shown in Table 3, and are equivalent to the expression notation we use in this paper. The high level description of the gesture interaction is the following: if the user is not in front of the screen, the application does not track his/her movements. When the user is in front of the screen, s/he can highlight one of the recipe types, which can be selected by a grab gesture (closing the hand).

The interaction is a *sequence* of different gestures, which starts with the user that standing in front of the screen (the screen front gesture, from line 7 to line 13 in Table 3). Such constraint is modelled checking the position of the shoulder points, which have to be almost parallel to the sensor plane on the depth axis. The constraint is computed using a C# method (*screenFront*) that is referenced by the value of the *Accept* attribute and it is defined in the code-behind file associated to a XAML specification.

When this gesture is completed, the user needs to be aware that the application is tracking his/her position, therefore the completion method associated to the gesture changes the message on the label at the bottom of the UI in Figure 1, setting its text to "Tracking user…" with a green background. The definition of this behaviour is again in the code-behind file, and it is linked with the gesture declaration through the *Method* attribute in the *change.completed* tag (from line 9 to 12 in Table 3). The method name in this case is *screenFront_Completed*.

Once this gesture is completed, it is possible to interact with the screen, and the grab gesture implements the selection of the recipe type. First, we listen iteratively to the change of the right hand position (the *Change* tag with *Feature="HandRight"* at line 17 in Table 3). Every time it is completed (read the user moves the hand), the *moveHand_Completed* method is executed. It updates the currently highlighted recipe type (the one with the red border in Figure 1).

After that, the recognition iteration is interrupted in two cases. The first one is when the user closes the right hand (the *Change* at line 24 in Table 3), and the method *rightHandClosed_Completed* handles the completion of the grab gesture, changing the current presentation. The second case is when the user goes away and s/he is not in front of the screen anymore (line 33 in Table 3). This situation is modelled symmetrically with respect to the gesture at line 7, the only difference is the *Accepts* method (*notScreenFront*), which is exactly the logical negation of *ScreenFront*. In both cases, the interruption is modelled using a *disabling,* declared respectively by the inner and the outer *Disabling* tags (line 14 and 16 in Table 3).

In summary, for creating a gestural interface with GestIT is sufficient to:

1. Create the UI view
2. Define the gestures associated to a view (in the same file), composing declaratively existing gestures or creating new ones starting from ground-terms.
3. Provide the methods for calculating the predicates associated to the specified gestures in the code-behind file (if any)
4. Provide the UI behaviour associated to the gesture completion

In the following sections we discuss how such organization solves the problems that are the topic of this paper.

Granularity Problem

The granularity problem derives from the modelling of complex gestures with a single event notification when it completes. Due to the time duration of the interaction gestures, it is usually needed to provide intermediate feedback during the performance, with the consequent need to split the complex gesture in smaller parts.

In order to show the impact of such problem even for simple interactions, here we focus on two specific hand gestures we exploited in the touchless recipe browser: the first one is a simple hand grab, which is used in the first and the second presentation for selecting an object. The second one is a hand-drag gesture we used for controlling the recipe preparation video: the user grabs the knob of the video timeline and then it moves is back and forth before "releasing" it by simply opening the hand.

```
<TabItem x:Name="recipeType">
 <Grid Background="#FF92BCED">
  <!-- gesture definition -->
   <g:GestureDefinition x:Name="moveSelection" >
     <g:Sequence Iterative="True">
       <!-- turn (front of the screen) -->
        <g:Change Feature="ShoulderLeft"
             Accepts="screenFront">
           <g:Change.Completed>
             <g:Handler
                method="screenFront_Completed"/>
         </g:Change.Completed>
        </g:Change>
        <g:Disabling>
        <!-- grab gesture  -->
          <g:Disabling Iterative="True">
            <g:Change Feature="HandRight"
                Iterative="True">
             <g:Change.Completed>
               <g:Handler
                method="moveHand_Completed" />
             </g:Change.Completed>
            </g:Change>
            <g:Change Feature="OpenRightHand"
                      Accepts="rightHandClosed">
              <g:Change.Completed>
                <g:Handler
            method="rightHandClosed_Completed"/>
              </g:Change.Completed>
           </g:Change>
         </g:Disabling>
       <!-- turn (not in front of the screen) -->
          <g:Change Feature="ShoulderLeft"
                    Accepts="notScreenFront">
            <g:Change.Completed>
              <g:Handler
              method="notScreenFront_Completed"/>
            </g:Change.Completed>
          </g:Change>
        </g:Disabling>
     </g:Sequence>
  </g:GestureDefinition>
 <!-- view definition -->
   <ui:FisheyePage x:Name="heading" />
    <kt:KinectSensorChooserUI
       Name="kinectSensorChooser1" />
  </Grid>
</TabItem>
```

Table 3 UI view and gesture definition of the recipe selection presentation

Table 4 shows how it is possible to model such gestures with GestIT. The grab gesture is composed by an iteration of the hand movement (mH_r^*), which is disabled by a change on the feature that tracks the opened or closed status of the hand (cH_r in the expression). We force the recognition only of a hand closure specifying the *closed* predicate, which accepts only changes from opened to closed. The grab gesture is a prefix for the drag one. Indeed, it is defined by a grab gesture followed in sequence by an iterative movement of the hand, disabled again by a change on the hand status, this time from opened to closed (modelled by the *open* predicate).

Grab	$mH_r^* [> cH_r[closed]$
Drag	$Grab \gg Release$ $Release = mH_r^*[> cH_r[open]$

Table 4: Grab and Drag hand gestures definition using GestIT. The expressions consider only the right-hand, the definition of the same gestures for the left hand is symmetric.

With GestIT it is possible to reuse the definition of the grab gesture for defining the drag one, as it is shown in Table 4. However, the possibility to compose gestures with a set of operators does not guarantee the reusability of the definition. Indeed, even in this simple example, the programmer needs a fine-grained control not only on the gesture itself, but also on it subparts. In the first two screens of the recipe browser application the grab gesture is exploited for an object selection, and the user has to be aware of which object s/he is currently pointing. Therefore, there is the need to provide intermediate feedback during the grab gesture execution. This is supported in the application exploiting the fact that GestIT notifies the completion of the gesture sub-parts. With this mechanism, the application receives a notification when each time mH_r is completed, highlighting the pointed object. The handler associated to the completion of the entire gesture performs the recipe selection and the presentation change. While performing the drag gesture, there is no need to attach a handler to the hand movement in the grab part, but it is sufficient to specify that the position in the video stream is changing after the grab completion, and to update it during the movement of the hand in the release part of the gesture.

It should be clear now how the declarative and compositional pattern offered by GestIT solves the granularity problem: the application developer is not bound to receiving a single notification when the whole gesture is completed. If needed, s/he is able to attach the behaviour also to the gesture sub-parts, handling them at the desired level of granularity.

Spaghetti Code Problem

The previous example may be used also for showing how to address the problem of having the gesture recognition code spread in many places (spaghetti code problem). Indeed, the declarative and compositional approach to the gesture definition allows the developer to separate the temporal sequencing aspect from the UI behaviour while defining a gesture. This allows maintaining the gesture recognition code isolated in a single place.

In the example, the recognition code corresponds to the declaration of the gesture expression. The handlers define the UI behaviour, but they are not part of the recognition code, since they are simply attached to the run-time notification of the gesture completion (or its sub-parts). In this way, it is not only possible to isolate the recognition code into a single application, but it is also possible to provide a library of complex gesture definitions, which may

be reused in different scenarios, maintaining the possibility to attach the UI behaviour at the desired level of granularity. In this particular example, it would be possible to model the entire interaction instantiating a single complex gesture. Indeed, the *Grab* and the *Release* gestures differ only for the predicate on the change of the hand status feature. Therefore, it is possible to define with GestIT a complex gesture that is parametric with respect to this predicate.

HandStatus	$HandStatus[p] = mH_r^* \; [> \; cH_r[p]$
Grab	$HandStatus[closed]$
Drag	$HandStatus[closed] \gg HandStatus[open]$

Table 5 Grab and Drag gestures defined using a single parametric complex gesture.

Table 5 shows a different definition of the gestures in Table 4, which demonstrates the level of flexibility in the factorization of the gesture recognition code in the proposed framework.

Selection Ambiguity Problem

In this section, we show how the problem of possible ambiguities that may arise when composing gestures is handled in GestIT. We exemplify the problem through a simple 3D viewer application [20]. The interaction with the 3D model is the following: the user can change the camera position performing a "grabbing" the model gesture with a single hand and moving it, while it is possible to rotate the model executing the same gesture with both hands. The complete definition is shown in Table 6. For the sake of simplicity we omit the part related to the left hand in the *Move* definition, but the point we are going to discuss is symmetrically valid also for the left hand.

$$
\begin{aligned}
&Move \; [] \; Rotate \\
&Move = cH_r[closed] \gg (mH_r^* \; [> \; cH_r[open]) \\
&Rotate = \; (cH_r[closed] || cH_l[closed]) \gg \\
&\qquad ((mH_r[d] || mH_l[d])^* [> \\
&\qquad (cH_r[open] || cH_l[closed]))
\end{aligned}
$$

Table 6: Gesture definition for the 3D viewer application

The *Move* and the *Rotate* gestures are composed through a choice operator but, as it is possible to see in the definition, both gestures start with $cH_r[closed]$. Therefore it is not possible to perform the selection immediately after the recognition of the first ground term, but the recognition engine needs at least one "lookahead" term, and the selection has to be postponed to the next event raised from the device. However, the two instances of $cH_r[closed]$ may have different handlers attached to the completion event, which should be executed in the meantime.

In general it is possible that, when composing a set of different gestures through the choice operator, two or more gestures have a common prefix, which does not allow an immediate choice among them. We identified three possible ways for addressing this problem. The different solutions have an impact on the recognition behaviour while traversing the prefix.

The first solution is the one proposed in [11], where the authors define an algorithm for extracting the prefix at design time. After having identified it, it is possible to apply a factorization process to the gesture definition expression, removing the ambiguity. This solution has the advantage that, since there is no ambiguity anymore, the recognition engine is always able perform the selection among the gestures immediately. The main drawback is that it breaks the compositional approach: after the factorization the two gesture definitions are merged and it is difficult for the designer to clearly identify them in the resulting expression. This leads to a lack of reusability of the resulting definition.

The second possible solution is again to calculate the common prefix at design time, without changing the gesture definition. In this case, the recognition support is provided with both the gesture definition and the identified prefix. During the selection phase at runtime, the support buffers the raw device events until only one among the possible gestures can be selected according to the pre-calculated prefix, and then flushes the buffer considering only the selected gesture. This approach has the advantage of maintaining the compositional approach, while selecting the exact match for the gestures in choice: the runtime support suspends the selection until it receives the minimum number of events for identifying the correct gesture to choose. Once the gesture has been selected, the application receives the notification of the buffered events. The latter is the main drawback of this approach: the buffering causes a delay on the recognition that is reflected on the possibility to provide intermediate feedback while performing the common prefix gesture. Another drawback is that the common prefix has to be calculated at design time, which may need an exponential procedure for enumerating all the possible recognizable event sequences, which are needed for extracting the common prefix. For instance, an order independence expression with n operands in GestIT recognizes $n!$ event sequences, since we should consider that the operands can be performed in any order.

The third solution is based on a best effort approach, and is the one implemented by GestIT. When two or more expressions are connected with a choice operand, the recognition support executes them as if they were in parallel. If the user correctly performed one of the gestures in choice, when the parallel recognition passes the common prefix only one among the operands can further continue in the recognition process. At this point the choice is performed and only one gesture is successfully recognized, and the support stops trying to recognize the others. This approach solves the buffering delay problem of the previous solution, since the effects of the gestures contained into the

common prefix is immediately visible for the user. However, in this case the recognition support notified the recognition of the gestures included in the common prefix of all the operands involved in the choice. Consequently, the UI showed the effects associated to all of them, while only the ones related to the selected gesture should be visible. In order to have a correct behaviour, we need a mechanism to *compensate* the changes made by the gestures that were not selected by the recognition support, which means to revert the effects they had on the UI. Such mechanism can be supported through a notification signalling that the recognition of a gesture (to all gestures (ground term or complex) has been interrupted. In this way it is possible for the developer to specify how to compensate the undesired changes. This is the main drawback for this solution: the developer is responsible of handling the compensating actions.

In order to better explain how this solution works, we present a small example of compensation. We consider the gesture model in Table 6, which allows the user to move and to rotate a 3D model. The UI provides intermediate feedback during the gesture execution in the following way: a four-heads arrow while the camera position is changing, and a circular arrow while the user is rotating the model.

We suppose in our example that the user performs the grab gesture with both hands and we describe the behaviour of the recognition support during the recognition of the common prefix (in this case $cH_r[closed]$) and after the gesture selection has been performed. The common prefix handling is depicted in Figure 2: the upper part represents the stream of updates that comes from the device, the black arrow highlights the one that is under elaboration. The central part shows the gesture expression represented as a tree, with the ground terms that can be recognized immediately highlighted in black (we do not show the predicates associated to the ground terms, since for this example we suppose that they are always verified). Some tree nodes are associated to rectangular and circular badges, which represent respectively the completion and the compensation behaviour. Such handlers are external with respect to the gesture description and are defined by the developer. The lower part shows the effects on the UI of the gesture recognition. The left part depicts the UI before the recognition, the middle part shows the intermediate effects, while the right one shows the resulting state after the recognition.

During the recognition of the common prefix, the support behaves as follows: after receiving the update coming from the device, the support executes the two instances of cH_r, highlighted by the black arrows in Figure 2, central part. Since the leftmost one has an associated completion handler (the A rectangular badge), the recognition support executes it. Therefore the UI changes its state and an arrow is shown above the 3D model (Figure 2, lower part). After that, the expression state changes (two ground terms have been recognized) and we have the situation depicted in Figure 3: the ground terms with a grey background have been completed, therefore the ground terms that may be recognized at this step are mH_r or cH_l. Since the next device update we are considering is cH_l (Figure 3, upper part), the recognition support is now able to perform the selection of the right-hand part of the expression tree, while the left-hand part cannot be further executed.

Therefore, the latter needs compensation, which consists of invoking the handlers associated to all the expressions previously completed (cH_r). In our example this corresponds to the execution of the handler identified with the B circular badge, which hides the four-heads arrow. After that, it is possible to continue with recognition of the gesture: the cH_l ground term in the right-hand part of the expression is completed and also the parallel expression highlighted with a black arrow in Figure 3. Consequently the recognition support executes the completion handler represented with the C rectangular badge, which shows the circular arrow for providing the intermediate feedback during the model rotation, and the gesture recognition continues taking into account only the *Rotate* gesture. The effects of the handlers on the UI for this step are summarized by the lower part of Figure 3: before the recognition of the ground term it was visible on the UI the four-head arrow, which has been hidden by the B compensation handler. The C completion handler instead showed the circular arrow that determines the state of the UI after the ground term recognition.

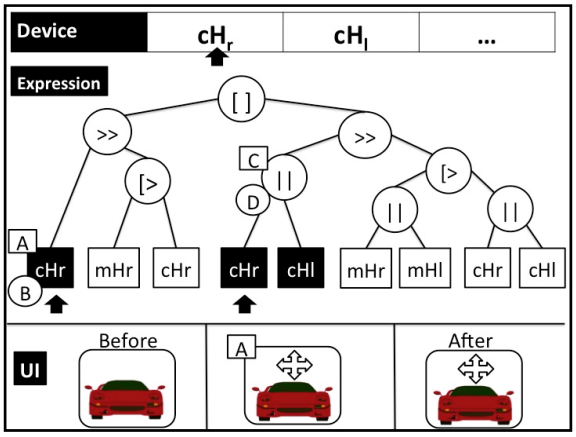


Figure 2: Example of common prefix handling in for the choice operator (part 1).

From a theoretical point of view, the proposed solution considers the set of gestures in choice as instances of long-running transactions [6], but in this case the components involved are not distributed. In case of failure of such kind of transactions, it is not possible in general to restore the initial state, as happens with the effects on the UI of the gestures that are not selected by the choice. Instead, a compensation process is provided, which handles the return to a consistent state. There is a large literature on how to

manage long-running transactions, in [3] the authors provide a good survey on the topic.

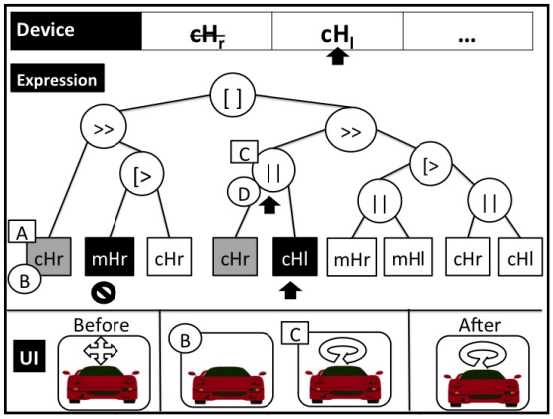


Figure 3: Example of common prefix handling in for the choice operator (part 2).

CROSS PLATFORM GESTURE MODELLING

Since the compositional gesturing model definition is based on a set of building blocks (ground terms), connected through a set of well-defined composition operators, it is possible to create interfaces that share the same gesture definition across different recognition platforms finding a meaningful translation of the source platform ground terms towards the target one. This opens the possibility to reuse the gesture definition not only for different applications that exploit the same recognition device but also, if the interaction provided still makes sense, with different devices that have different recognition capabilities. In order to explain how such reuse is possible, we report here on a first experiment we conducted with the two platforms supported by GestIT: multitouch and full-body.

$$Pan\ [\ ]Pinch$$
$$Pan = Start_1 \gg Move_1^* \ [> \ End_1$$
$$Pinch = (Start_1 |=| Start_2) \gg (Move_1^* \ | \ | Move_2^*) \ [> \ (End_1 |=| End_2)$$

Table 7: Simple drawing canvas gesture modelling

We started from a simple drawing canvas application for iPhone, which supported the pan gesture for drawing and the pinch gesture for zooming. Such gestures were connected through the choice operator, as defined in Table 7. Notice how it is easy to support the zooming feature while drawing by simply changing the choice operator to the parallel one, without any additional effort for the developer.

The UI behaviour associated to the gesture definition can be summarized as follows:

- To the *Move* block of the pan gesture we associated an event handler that draws a line from the previous touch position to the current one.
- To each one of the *Move* blocks of the pinch gesture, we associated an event handler that computes the difference between the previous and the current distance between the two touches. If it is increased, the canvas zooms in the view, otherwise it zooms out the view accordingly

In order to create a full-body version of the same application, it is not possible to reuse directly the gesture definition, because concepts as pan, pinch, touch etc. do not have any meaning in such platform. However, having a precise definition of the gesture also allows us to define precisely new concepts. In our case, what is missing is a precise definition of what a touch start, a touch move and a touch end are. If we add a precise definition of these concepts, all the gestures that have been constructed starting from such building blocks will be defined consequently. One simple idea is to associate a point that represents a finger position on the iPhone to the position of one hand with the Kinect (therefore, the maximum number of touch points is two). In addition, we have to define a criterion for distinguishing when the touch starts and when it ends. A simple way is to rely on the depth value of the position of a given hand: if it is under a certain threshold, we can consider that the user is "touching" our virtual screen, otherwise we do not consider the current hand position as a touch.

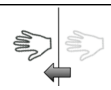
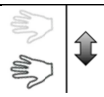
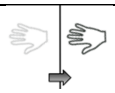
Multitouch Ground Term	Interaction
$Start_1 = r[z_r(t-1) > k \wedge z_r(t) \leq k]$ $Start_2 = l[z_l(t-1) > k \wedge z_l(t) \leq k]$	
$Move_1 = r[z_r(t-1) \leq k \wedge z_r(t) \leq k]$ $Move_2 = l[z_l(t-1) \leq k \wedge z_l(t) \leq k]$	
$End_1 = r[z_r(t-1) \leq k \wedge z_r(t) > k]$ $End_2 = l[z_l(t-1) \leq k \wedge z_l(t) > k]$	

Table 8: Mapping of the multitouch ground terms to the full-body platform

More precisely, we need to define the multitouch basic gestures according to the 3D position of the left and right hand, indicated respectively as $l = (x_l, y_l, z_l)$ and $r = (x_r, y_r, z_r)$. Moreover, we have to define a plane, which represents the depth barrier for the touch emulation, as $T_p = (x, y, k)$ where k is a constant. The complete definition can be found in Table 8.

It is worth pointing out that, even if we used such definition for a quite "extreme" change of platform, the redefinition of the ground term allows us to support with the Kinect platform all the multitouch gestures that involve no more than two fingers, which are the large majority of those used in such kind of applications. Obviously, from the interaction design point of view it may be a bad idea to port multitouch gestures to the full body gesture platform directly, and the example should be considered only as a proof of concept. However, such kind of approach may be used for those devices that are exploited for recognizing gestures in similar settings. For instance, it can be useful for designing applications that recognize the same full body gestures with a remote or a depth camera-based optical device. In this case,

having such kind of homomorphism may reduce the complexity in supporting different devices.

CONCLUSION AND FUTURE WORK

The spread of gesture interfaces both in mobile devices, in game settings and more recently in smart environments is pushing for solving the problem of having a different programming paradigm, with respect to the single-event notification for describing gestures. Declarative and compositional approaches for gesture definition represent a step further towards such a new model, solving the single-event granularity problem and providing a separation of concerns (the temporal sequence definition is separated from the behaviour), which allows a more understandable and maintainable code. In addition, we discussed the selection ambiguity problem, which affects the composition of gestures that have a common prefix through a choice operator. The recognition support has different possibilities for dealing with the uncertainty in the selection while performing this common prefix. We discussed the different solutions using GestIT as a sample framework and we demonstrated that it is more expressive than other libraries in literature.

In the future, we plan to enhance the framework adding the support for more platform and devices (e.g. remotes). In addition we will exploit the declarative approach for identifying gestures that are not used directly for the interaction (*posturing*) but that may be used in order to detect the user's emotional status.

REFERENCES

1. Android Developer, Responding to Touch Events. http://developer.android.com/training/graphics/opengl/touch.html, retrieved 12-10-2012.
2. Apple Inc., Event Handling Guide for iOS. http://developer.apple.com/library/ios/navigation/, retrieved 12-10-2012.
3. Colombo, C., Pace, G. Recovery within Long Running Transaction, ACM Computing Surveys 45 (3), 2013 (accepted paper).
4. Echtler, F., Butz, A. GISpL: Gestures Made Easy. *In Proc. of TEI '12*, pp. 233-240, ACM, (2012).
5. Meyer, A. S., Gesture Recognition. http://wiki.nuigroup.com/Gesture_Recognition, retrieved 12-10-2012
6. Garcia-Molina, H., Gawlick, D., Klein, J., Kleissner, K., Salem, K., Modelling Long-Running Activities as Nested Sagas. IEEE bulletin of the Technical Committee on Data Engineering, 14 (1), 1991.
7. Gibbon, D., Gut, U., Hell, B., Looks, K., Thies, A., and Trippel, T. A computational model of arm gestures in conversation. Proc. Eurospeech 2003, ISCA (2003), 813–816.
8. Hoste, L., Dumas, B., Signer, B. Mudra: a unified multimodal interaction framework. *In Proc. of ICMI '11*, pp. 97-104, ACM, (2011).
9. Kammer, D.,Wojdziak, J., Keck, M., and Taranko, S. Towards a formalization of multi-touch gestures. Proc. ITS 2010, ACM, (2010), 49–58.
10. Khandkar, S. H., Maurer, F. A domain specific language to define gestures for multi-touch applications. *In Proc. of DCM'10 Workshop*, Article No. 2, ACM, (2010).
11. Kin, K., Hartmann B., DeRose. T., and Agrawala, M., Proton: multitouch gestures as regular expressions. In CHI 2012 (Austin, Texas, U.S. May 2012)., 2885-2894
12. Kin, K., Hartmann, B., DeRose, T., Agrawala, M.. Proton++: a customizable declarative multitouch framework. *In Proc of UIST 2012*. ACM, New York, NY, USA, 477-486.
13. Microsoft, Kinect for Windows SDK, http://www.microsoft.com/en-us/kinectforwindows/, retrieved 12-10-2012
14. Brian, P., WiimoteLib, http://www.brianpeek.com/page/wiimotelib, retrieved 12-10-2012.
15. Paternò, F. Model-based design and evaluation of interactive applications. *Applied Computing*, Springer 2000
16. Primesense, NITE, http://www.primesense.com/en/nite, retrieved 12-10-2012
17. Scholliers, C., Hoste, L., Signer, B., De Meuter, W., Midas: a declarative multi-touch interaction framework. *In Proc. of TEI'11*, pp. 49–56, ACM, (2011)
18. Schwarz, J., Hudson, S. E., Makoff, J., Wilson, A. D. A framework for robust and flexible handling of inputs with uncertainty. *In Proc. of UIST 2010*, pp. 47-56, ACM, (2010)
19. Sourceforge, WiiLib, http://sourceforge.net/projects/wiilib/, retrieved 12-10-2012
20. Spano, L.D., Cisternino, A., Paternò F., A Compositional Model for Gesture Definition, *In Proc. of HCSE*, LNCS, 7623, pp. 34-52, Springer, (2012)
21. Spano, L.D., Developing Touchless Interfaces with GestIT, *In Proc. of AMI 2012*, LNCS, 7683, pp. 433-438, Springer (2012).